Adaptive Voxel Subdivision for Ray Tracing

David Jevans and Brian Wyvill
University of Calgary
Department of Computer Science
2500 University Drive N.W.
Calgary, Alberta, Canada, T2N 1N4

Abstract

Although regular subdivision has been shown to be efficient at ray tracing scenes where objects are evenly distributed, such algorithms perform poorly when objects are concentrated in a small number of voxels. In this paper, a method is presented where voxels in a regular grid are examined and recursively subdivided depending on object density. This integration of regular and adaptive spatial subdivision methods allows images consisting of large regularly distributed objects and small dense objects to be ray traced efficiently. The parameters controlling the coarseness of the voxel grid, depth of adaptive subdivision trees, and maximum number of polygons per voxel are varied and their effects on execution time, subdivision time, and memory use are measured.

Key Words: ray tracing, spatial subdivision, voxel, octree.

1 Previous Work

Speeding up execution time has been a major issue in the development of ray tracing since its inception. A naive ray tracer which tests every ray against every object in a scene for possible intersection will spend most of its time testing rays against objects with which they do not intersect. Rubin and Whitted [Rubin 80] determined that between 75% and 95% of ray tracing time was spent performing such tests. Much research has been done to reduce the number of intersection tests required to ray trace a scene. The major contributions follow.

Rubin and Whitted used hierarchical bounding rectangular parallelepipeds around the objects in a scene to reduce the number of intersection tests required to trace a ray through it [Rubin 80]. Rays are tested for intersection with the bounding volumes and are intersected with child volumes if they intersect the parent. Intersection tests are performed only for the objects bounded by leaf volumes with which a ray intersects. These hierarchical volumes must be constructed by hand for each scene, a time consuming process not suitable for animation.

Vatti subdivided object space with a grid of equally sized cubes (*voxels*) [Vatti 84]. Objects in a scene are sorted into the grid and rays are traversed through it. Intersection tests are only performed for the objects which lie inside voxels through which a ray passes.

Glassner and Kaplan subdivide space adaptively with an octree rather than a regular voxel grid [Glassner 84] [Kaplan 85]. This allows scenes with varying object densities to be rendered more efficiently than with a regular grid. Algorithms for rapid octree traversal are presented, whereas a similar algorithm for voxel traversal is lacking in Vatti's work.

Fujimoto compares voxel and octree space subdivision methods for ray tracing polygonal objects and implicit surfaces (*meta balls*), and presents an incremental 3DDA algorithm for rapidly traversing rays through regular voxel grids [Fujimoto 86].

Kay and Kajiya present a variation on the Rubin and Whitted hierarchical bounding volume algorithm [Kay 86]. Instead of rectangular parallelepipeds they bound their objects with arbitrarily tight fitting volumes which can quickly be intersected with rays. A comparison with Glassner's algorithm is presented with results showing that the bounding volume algorithm is two to three times faster than the octree algorithm.

Cleary and Wyvill perform an analysis of regular voxel subdivision and present a voxel skipping algorithm which is superior to the Fujimoto algorithm [Cleary 88]. Amanatides [Amanatides 87] independently develops a similar algorithm but does not optimise it to the extent of the Cleary algorithm.

Snyder and Barr combine regular subdivision and hierarchical bounding volumes to ray trace tessellated surfaces composed of millions of objects [Snyder 87]. A voxel skipping algorithm similar to the Cleary method is used to traverse the voxel grid. This is an example of a hybrid of two methods which results in a faster algorithm than either one alone. A similar approach was taken by Jevans and Wyvill who combined regular voxel subdivision and octree subdivision to ray trace implicit surfaces [Jevans 88].

```
begin subdivide_voxel
{
   recursion_depth = recursion_depth + 1
   if recursion_depth < MAXDEPTH
      for each voxel
         if the number of polygons in the voxel > MAXP {
            subdivide the voxel into N^3 sub-voxels
            call subdivide_voxel with the sub-voxel grid
         }
}

where:
   MAXP is the maximum number of polygons per voxel.
   N is the number of sub-voxels on a grid side.
   MAXDEPTH is the maximum tree depth.
```

Figure 1: Recursive subdivision algorithm.

Arvo and Kirk propose a five dimensional space subdivision technique that makes use of the direction of rays as well as the areas of space through which they travel [Arvo 87].

2 Assumptions of Other Algorithms

Adaptive spatial methods, such as the octree, are based on the observation that, in typical scenes, large areas of space are often empty while areas which contain objects of interest are often quite densely populated. These algorithms typically require more computation than regular subdivision techniques [Fujimoto 86] since traversal of a ray through a scene requires many vertical octree traversals. These contribute significantly to the execution time but get a ray no further along its path through a scene.

Hierarchical object grouping methods assume that objects are constructed in a hierarchical manner or can be decomposed in such a manner. They often require hand tuning to sort objects into hierarchies. This can be very difficult when rendering objects which are composed of polygons generated in an arbitrary order. Such objects are often the result of special purpose modeling tools, scientific data, or polygonal representations of functionally defined surfaces.

Regular subdivision methods assume that objects are uniformly distributed throughout space and that one can afford to subdivide space finely enough to have an optimal number of polygons per voxel even in very dense areas. These methods can fail when rendering scenes which include small objects composed of many thousands of polygons, since there may not be sufficient memory available to adequately subdivide the scene in these areas. The scenes that are rendered by the *Graphicsland* group at the University of Calgary often fall into this category. They are usually frames from animation sequences that have large regularly distributed background objects and a small number of dense foreground objects which can comprise up to ninety percent of the polygons in the scene.

This paper describes a generalisation of the octree method that extends adaptive space subdivision to orders greater than 2^3. The method employs both octree and regular subdivision techniques to efficiently render scenes without making assumptions about them.

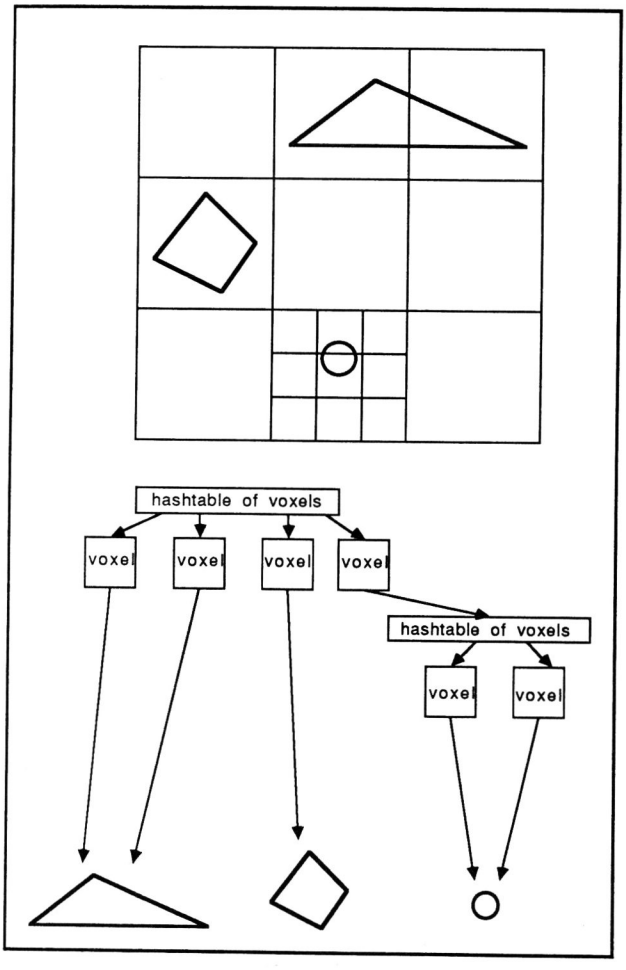


Figure 2: A 2D Adaptively Subdivided Scene.

3 Adaptive Subdivision

The subdivision algorithm proceeds in a similar manner to an octree insertion algorithm except that the order is greater than or equal to 2^3. All polygons are initially read into the program and inserted into a single voxel. Figure 1 illustrates the recursive subdivision algorithm.

Each voxel structure can contain either a list of objects inside that voxel if it is a leaf node, or a voxel grid subdividing it further if it is an internal node. Octree methods usually use 8 pointers or a table of 8 indices to represent the 2^3 subdivisions of a given node. As the order of a node grows beyond 2^3 it becomes impractical to store a pointer for each voxel in a sub-grid. Instead, only references to non-empty voxels are stored.

A hashtable of buckets is maintained in each subdivided voxel structure (see Figure 2). Each bucket points to a non-empty child voxel which may in turn be subdivided. In order to speed up voxel lookup, a bit table which is large enough to contain one bit per voxel is stored with each hashtable. To determine if a particular child voxel is non-empty the bit table is consulted. A non-zero bit table entry indicates that the voxel is non-empty and the hashtable must be indexed and searched for the voxel data structure.

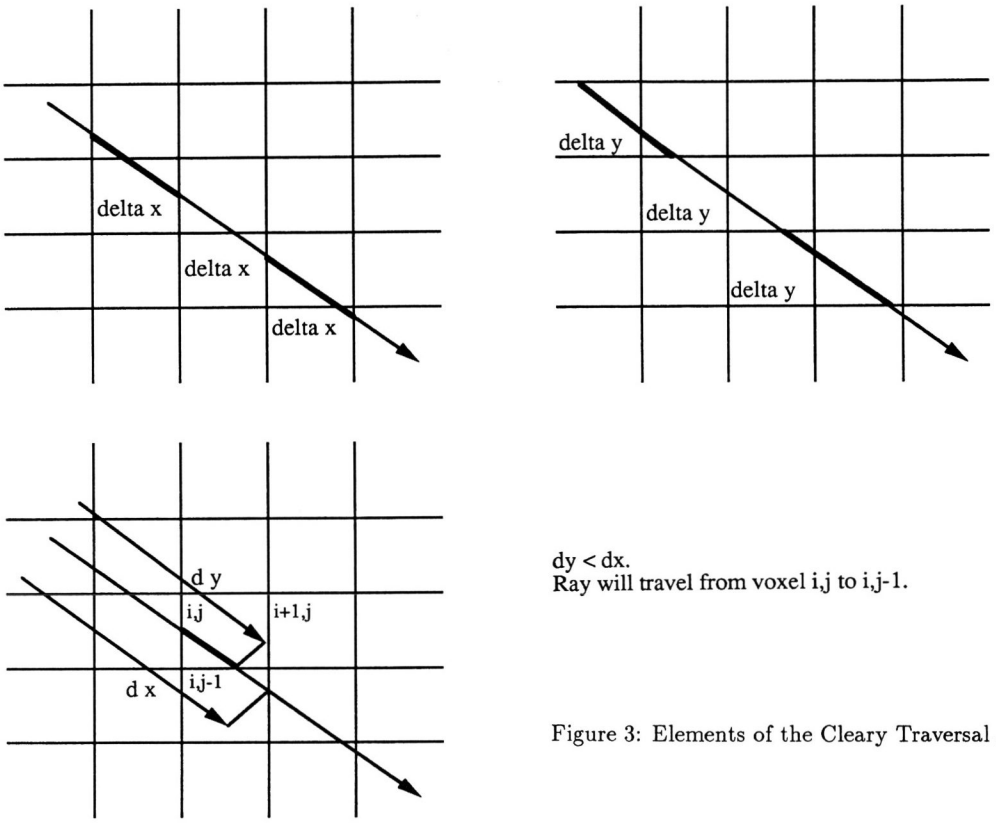


Figure 3: Elements of the Cleary Traversal Algorithm.

Pearce showed that a one dimensional space results in faster voxel traversal and easier to generate hashtable and bit table indices. In this implementation the one dimensional voxel grid traversal algorithm and hashtables presented in [Pearce 87] are used.

4 Voxel Traversal

4.1 Horizontal Traversal

The Cleary voxel traversal algorithm is used and a summary follows for completeness [Cleary 88]. Consider Figure 3 which shows the elements of this algorithm for the 2D case. At the start of each ray the following values are initialised: $d[3]$, the distances along the ray to the nearest voxel boundary in the x, y, and z directions; *delta*[3], the distances along the ray between voxel boundaries in the x, y, and z directions; and the *small* variable, which indicates the closest voxel to the ray's origin. The voxel in which the ray originates is also found at this point.

The algorithm proceeds as outlined in Figure 4. The traversal iteration loop checks the state of the current voxel. If it is a non-empty leaf node, the ray is intersected with the objects that lie in the voxel. Should the ray intersect an object, the properties of the ray and the surface with which it intersects are passed to a shading routine. If the voxel is empty or the ray does not intersect any of the objects that lie inside it, then the next voxel is determined and the algorithm loops.

The next voxel into which the ray will travel is indicated by the *small* variable. $d[small]$ is incremented by positive or negative $delta[small]$, the new value for *small* is determined by examining the $d[i]$s, the ray's position in the voxel grid, indicated by the $index1D$ and $index3D[3]$ variables, is updated, and the algorithm loops.

In Pearce's implementation of Cleary's algorithm, bounding polygons are placed around the scene and, when a ray intersects a bounding polygon, the ray is known to have left the voxel grid. The hybrid algorithm presented here does not use this method since rays are traced through many small sub-voxel grids, and the overhead of finding a ray–polygon intersection each time a ray leaves a subdivided voxel would be large. Instead, a 3D index is used to determine when a ray leaves a voxel grid. A ray is determined to have crossed a sub-voxel boundary when an index becomes negative, or is incremented beyond $N - 1$.

The main difference between this and the Cleary and Pearce implementations is the termination test. There are two indices which are incremented to determine the next voxel. The variable $index1D$ indexes into the hash table of voxels. The variable $index3D[3]$ is used to determine if the ray has emerged from the N^3 divided voxel.

```
begin traverse_voxel_grid
{
   initialise d[3], delta[3], small, index3D[3],
              index1D, indexinc[3]
   while true {
      if voxel not empty{
         if not a leaf node{
            call traverse_voxel_grid with sub_grid
            if ray intersected an object
               return
         } else {
            intersect ray with all objects
            if ray intersected an object
               shade ray and return
         }
      }
      if dir[small] < 0.0 {
         index3D[small] -= 1
         index1D -= indexinc[small]
      } else {
         index3D[small] += 1
         index1D += indexinc[small]
      }
      if index3D[small] < 0 or index3D[small] >= N
         return
      d[small] += delta[small]
      if d[0] < d[1]
         small = 0
      else
         small = 1
      if d[2] < d[small]
         small = 2
   }
}
```

(array notation: *[0]* = x, *[1]* = y, *[2]* = z direction)
where:

- *d[3]* is the distance along the ray from its origin to the first voxel boundary.
- *small* is an index into *d[3]* indicating the closest voxel boundary (ie. the next voxel the ray will enter).
- *delta[3]* is the distance along the ray between two voxel boundaries.
- *dir[3]* is the direction vector of the ray.
- *index3D[3]* is the index of the ray in the 3D grid.
- *index1D* is the index of the ray in the 1D grid.
- *indexinc[3]* is the amount to increment *index1D* when the ray moves to a new voxel.

Figure 4: Pseudo-code for the Cleary Traversal Algorithm.

4.2 Vertical Traversal

When a ray is traversing a voxel grid and encounters a voxel which is not a leaf node, a downward traversal must be made into the subdivided voxel. First, the point at which the ray intersects the subdivided voxel grid must be determined. From this the initial sub-voxel in the child's voxel grid can be found. The new grid is then traversed by calling the horizontal traversal routine, **traverse_voxel_grid**, recursively.

Since the origin, slope, and distance along the ray are known, the intersection of the ray with the sub-voxel grid can be found with one division, one subtraction, two multiplications, and three additions (see Figure 5). A further optimisation can be had by noting that the *delta*[3] values can be scaled by the relative size differences between the parent and the child voxel grid instead of being recalculated.

```
temp = min[oldsmall] - ori[oldsmall]
scale = temp / dir[oldsmall]
for ( a = 0; a < 3; a++) {
   if(a != oldsmall)
      intersect[a] = ori[a] + scale * dir[a]
   else
      intersect[a] = ori[a] + temp
}
```

where:

- *min[3]* is the coordinates of the "lower left" corner of the current voxel grid (stored in the voxel data structure).
- *max[3]* is the coordinates of the "upper right" corner of the current voxel grid (stored in the voxel data structure).
- *oldsmall* is the previous value of the *small* variable.
- *ori[3]* is the ray's origin (its intersection with this voxel grid).
- *intersect[3]* is the ray's intersection with the sub-voxel grid and will be its new origin for downwards traversal.

Figure 5: Calculations for a vertical traversal.

Upward traversals do not require inverting these operations since the old values are stored at each recursive downward traversal call. When the traversal of a ray through a child's sub-voxels returns and has not intersected an object, the parent's horizontal voxel traversal continues where it left off. If a successful intersection is recorded, the current ray is terminated and secondary rays are spawned.

5 Ray–Object Intersection

When a ray enters a non-empty leaf voxel it must be checked against all objects in the voxel for intersection. Once this is performed for each object in the voxel, any intersection points must be sorted by distance along the ray to determine which intersection is the closest to the ray's origin.

To avoid rechecking a ray for intersection with an object that lies in two adjacent voxels, the result of the last ray–object intersection calculation is stored inside each object. Each ray is uniquely numbered and the intersection result is tagged with this number. When a ray is to be tested against an object, the ray's tag and the tag stored in the object are compared as in [Arnaldi 87]. If they match, then the result stored in the object is used instead of being recalculated. While this requires more memory than a simple list of objects, it avoids the problem of intersecting a ray with the same object more than once.

6 Subdivision Methods

The adaptive voxel mechanism presented in this paper lends itself to many subdivision schemes and can be used as a testbed for comparative studies.

No Subdivision. A naive ray tracer can be simulated by setting the maximum depth of the subdivision tree to 0. No subdivision overhead will be incurred.

Octree. A traditional octree data structure can be enforced by setting the size of the voxel grids to 2 on a side. Traversal of this data structure is not as efficient as the traversal mechanism presented by Glassner [Glassner 84], because the code to initialise variables for voxel traversal is unnecessary for traversal of a 2^3 grid.

Regular Voxel Subdivision. A voxel grid structure can be enforced by setting the maximum depth of the subdivision tree to 1. Traversal of this data structure is done with the Cleary voxel traversal mechanism and does not suffer from the overhead of vertical traversal.

Adaptive Voxel Subdivision - Fixed Resolution. This data structure is the result of setting the resolution of the voxel sub-grids to a constant value (i.e. constant N) before rendering. The problem with this method is that of determining a value for N. When ray tracing irregularly distributed scenes, this method performs better than a simple voxel grid, but does not adapt well to very large variations in object densities throughout a scene.

Adaptive Voxel Subdivision - Variable Resolution. Each time a sub-voxel grid is to be generated, the resolution of the grid (N) is determined by the number of objects that lie inside it. This method can adapt to variations in object density in a scene better than the fixed resolution method, although a means for determining N at each subdivision must be developed.

7 Subdivision On-The-Fly

Many previous subdivision methods require a scene to be subdivided in a pre-processing step. This approach is necessary for regular subdivision and is usually used in octree methods. Pre-processing the scene can be wasteful if rays are never traversed through large portions of it.

Our implementation of adaptive voxel subdivision does all subdivision on-the-fly. When a ray enters a voxel that has not been subdivided and requires it, the subdivision routine is called. This method only subdivides the voxels through which rays pass, and can greatly reduce subdivision time for complex scenes. On-the-fly subdivision also allows for garbage collection of least recently used voxels, enabling machines with limited amounts of memory to render very complex scenes.

8 Results

The ray tracer was tested empirically using several test scenes which represent typical images rendered at the University of Calgary. A series of images of randomly placed cubes created with differing distributions were also rendered.

All rendering was performed on a SUN 4/280 at a resolution of 512 by 512 pixels without anti-aliasing. For each image, N, MAXP, and MAXDEPTH (as defined in Figure 1) were varied and the cpu time (in minutes) and memory use (in megabytes) measured. Garbage collection was disabled so that the total amount of memory required could be measured. Fixed grid resolutions were used to simplify our initial reference measurements.

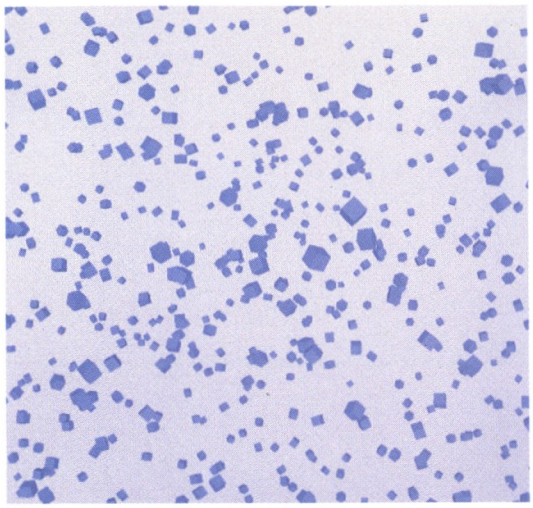
Note: timing results may suffer from a 2 to 3 percent error due to defficiencies in the SUN4 timing mechanisms. Table entries with a $*$ indicate that sufficient memory was not available to render the images without garbage collection.

Randomly distributed cubes. A series of test scenes were generated by distributing 1000 cubes in a 10 by 10 by 10 volume using four different distribution functions: negative exponential, normal, poisson, and uniform. As expected, regular subdivision ran faster and used less memory than the adaptive method, which in turn ran faster and used less memory than the octree method. However, these images do not represent the vast majority of scenes found in computer animation.

Teapot. A typical image in computer graphics, illustrating smooth shading, reflection, and shadowing. The teapot occupies most of the scene, allowing regular subdivision techniques to perform well on the image. This is not a typical animation scene as there is only a small and simple ground plane as a background.

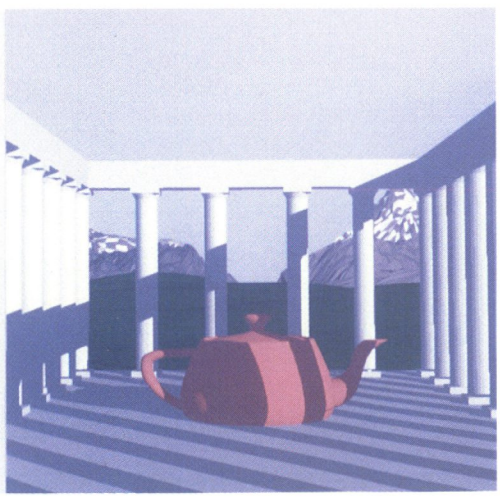
Temple with teapot. This scene is becoming more typical in computer animation. Large background objects and small, dense foreground objects make up the scene. This image illustrates the advantage of the adaptive subdivision technique, as it outperforms both regular subdivision and octree methods.

Lumpy. This scene from a new *Graphicsland* film, "Lumpy's Quest for Sev", exemplifies a typical animation sequence. The scene features a large, well distributed background, many smaller more complicated foreground objects, and several very small and complicated focal objects. There are a great many secondary rays involved in rendering this image since there are eight light sources which require shadow rays to be fired. Rendering this scene, the adaptive algorithm greatly outperforms both the octree and regular subdivision methods.



Uniform distribution. 6,000 polygons.
262,144 initial rays. 0 secondary rays. 262,144 total
1,293 minutes to render with no subdivision.

Teapot. 2049 polygons.
262,144 initial rays. 473,117 secondary rays. 735261 total
1,202 minutes to render with no subdivision.



Teapot in a temple. 44,479 polygons.
262,144 initial rays. 183,500 secondary rays. 445,644 total
16,446 minutes to render with no subdivision.

Lumpy. 5460 polygons.
262,144 initial rays. 1,457,101 secondary rays. 1,719,245 total
7,752 minutes to render with no subdivision.

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.5m	0.5m	0.5m	0.5m	0.5m
	Ren:	636.6m	303.6m	119.7m	92.2m	90.9m	90.9m
	Mem:	0.7Mb	0.9Mb	1.2Mb	1.6Mb	1.6Mb	1.6Mb
50	Sub:	0.4m	0.5m	0.5m	0.5m	0.5m	0.5m
	Ren:	649.2m	308.4m	110.5m	71.7m	70.8m	70.8m
	Mem:	0.7Mb	0.9Mb	1.3Mb	2.2Mb	2.4Mb	2.4Mb
10	Sub:	0.4m	0.5m	0.5m	0.5m	0.6m	0.6m
	Ren:	626.5m	300.6m	107.0m	60.3m	56.8m	56.8m
	Mem:	0.7Mb	0.9Mb	1.6Mb	4.1Mb	10.3Mb	13.8Mb

Table 1: Normal Distr. N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	63.8m	55.0m	55.5m	55.9m	55.5m	55.0m
	Mem:	0.7Mb	1.1Mb	1.1Mb	1.1Mb	1.1Mb	1.1Mb
50	Sub:	0.3m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	64.7m	36.7m	36.7m	36.3m	36.3m	36.3m
	Mem:	0.7Mb	2.2Mb	2.2Mb	2.2Mb	2.2Mb	2.2Mb
10	Sub:	0.3m	0.4m	0.5m	0.5m	0.5m	0.5m
	Ren:	64.2m	26.6m	27.1m	27.1m	26.6m	26.6m
	Mem:	0.7Mb	4.9Mb	5.5Mb	5.5Mb	5.5Mb	5.5Mb

Table 2: Normal Distr. N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.3m	0.3m	0.3m	0.3m	0.3m
	Ren:	25.3m	24.9m	25.8m	24.5m	24.5m	24.5m
	Mem:	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb
50	Sub:	0.3m	0.3m	0.3m	0.3m	0.3m	0.3m
	Ren:	24.5m	24.5m	24.5m	24.9m	24.5m	24.9m
	Mem:	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb
10	Sub:	0.3m	0.9m	0.9m	0.9m	0.9m	0.9m
	Ren:	24.9m	24.9m	24.9m	24.9m	24.9m	24.5m
	Mem:	0.8Mb	18.5Mb	18.5Mb	18.5Mb	18.5Mb	18.5Mb

Table 3: Normal Distr. N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.4m	0.4m	0.3m	0.3m	0.3m
	Ren:	15.3m	15.3m	15.3m	15.3m	15.7m	15.7m
	Mem:	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb
50	Sub:	0.3m	0.4m	0.3m	0.3m	0.3m	0.4m
	Ren:	15.7m	15.3m	15.3m	15.3m	15.3m	15.3m
	Mem:	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb
10	Sub:	0.4m	8.5m	8.6m	8.6m	8.4m	8.4m
	Ren:	15.3m	16.6m	16.6m	16.6m	16.6m	16.6m
	Mem:	1.0Mb	111.3Mb	111.3Mb	111.3Mb	111.3Mb	111.3Mb

Table 4: Normal Distr. N=40

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.5m	0.5m	0.6m	0.6m	0.6m
	Ren:	263.4m	69.5m	32.3m	28.4m	27.5m	27.1m
	Mem:	0.7Mb	0.9Mb	1.2Mb	1.7Mb	1.7Mb	1.7Mb
50	Sub:	0.4m	0.5m	0.5m	0.6m	0.6m	0.6m
	Ren:	261.3m	69.0m	27.1m	24.0m	22.7m	22.3m
	Mem:	0.7Mb	0.9Mb	1.3Mb	2.4Mb	3.1Mb	3.1Mb
10	Sub:	0.4m	0.5m	0.5m	0.6m	0.7m	0.7m
	Ren:	264.3m	66.8m	27.1m	20.5m	20.5m	21.0m
	Mem:	0.7Mb	0.9Mb	1.6Mb	4.2Mb	11.7Mb	19.6Mb

Table 5: Poisson Distr. N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.5m	0.5m	0.5m	0.5m	0.5m
	Ren:	15.3m	14.1m	13.6m	13.6m	14.4m	14.0m
	Mem:	0.8Mb	2.2Mb	2.2Mb	2.2Mb	2.2Mb	2.2Mb
50	Sub:	0.4m	0.5m	0.5m	0.5m	0.5m	0.6m
	Ren:	14.4m	11.4m	11.4m	12.2m	11.8m	12.2m
	Mem:	0.8Mb	4.0Mb	4.0Mb	4.0Mb	4.0Mb	4.0Mb
10	Sub:	0.4m	0.6m	1.1m	1.1m	1.1m	1.0m
	Ren:	15.7m	10.5m	10.1m	10.5m	10.5m	10.1m
	Mem:	0.8Mb	8.1Mb	9.3Mb	9.3Mb	9.3Mb	9.3Mb

Table 6: Poisson Distr. N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.4m	0.4m	0.4m	0.3m	0.4m
	Ren:	9.2m	8.7m	9.2m	8.7m	8.7m	8.3m
	Mem:	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb
50	Sub:	0.4m	0.9m	0.9m	0.9m	0.9m	0.9m
	Ren:	8.3m	7.9m	7.9m	8.3m	8.3m	8.7m
	Mem:	0.9Mb	7.4Mb	7.4Mb	7.4Mb	7.4Mb	7.4Mb
10	Sub:	0.4m	6.0m	5.9m	5.9m	5.8m	5.7m
	Ren:	10.1m	9.6m	8.7m	9.2m	8.3m	8.3m
	Mem:	0.9Mb	94.2Mb	94.2Mb	94.2Mb	94.2Mb	94.2Mb

Table 7: Poisson Distr. N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	7.4m	7.4m	7.9m	7.4m	7.4m	7.9m
	Mem:	1.3Mb	1.3Mb	1.3Mb	1.3Mb	1.3Mb	1.3Mb
50	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	7.4m	7.4m	7.4m	7.9m	7.9m	8.3m
	Mem:	1.3Mb	1.3Mb	1.3Mb	1.3Mb	1.3Mb	1.3Mb
10	Sub:	0.4m	*	*	*	*	*
	Ren:	8.3m	*	*	*	*	*
	Mem:	1.3Mb	*	*	*	*	*

Table 8: Poisson Distr. N=40

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	528.6m	158.6m	89.1m	91.3m	90.0m	89.6m
	Mem:	0.7Mb	0.9Mb	1.5Mb	1.5Mb	1.5Mb	1.5Mb
50	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	532.6m	155.5m	70.3m	67.7m	69.0m	67.3m
	Mem:	0.7Mb	0.9Mb	2.0Mb	2.0Mb	2.0Mb	2.0Mb
10	Sub:	0.4m	0.4m	0.4m	0.5m	0.5m	0.5m
	Ren:	525.6m	152.9m	63.8m	58.5m	59.0m	58.5m
	Mem:	0.7Mb	0.9Mb	2.0Mb	7.0Mb	8.8Mb	9.2Mb

Table 9: Uniform Distr. N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.3m	0.3m	0.3m	0.3m	0.3m
	Ren:	33.6m	33.6m	33.6m	33.6m	33.6m	33.6m
	Mem:	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb
50	Sub:	0.3m	0.3m	0.3m	0.3m	0.3m	0.3m
	Ren:	38.0m	38.0m	38.0m	38.0m	38.0m	38.5m
	Mem:	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb	0.8Mb
10	Sub:	0.3m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	38.0m	35.0m	35.0m	35.0m	35.4m	35.0m
	Mem:	0.8Mb	5.9Mb	5.9Mb	5.9Mb	5.9Mb	5.9Mb

Table 10: Uniform Distr. N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.4m	0.4m	0.3m	0.4m	0.4m
	Ren:	17.0m	17.0m	17.9m	17.5m	17.0m	17.0m
	Mem:	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb
50	Sub:	0.4m	0.3m	0.4m	0.3m	0.3m	0.3m
	Ren:	17.0m	17.5m	17.0m	17.5m	17.5m	17.0m
	Mem:	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb
10	Sub:	0.3m	0.5m	0.5m	0.5m	0.5m	0.5m
	Ren:	14.8m	15.3m	15.3m	15.3m	15.3m	15.3m
	Mem:	0.9Mb	4.0Mb	4.0Mb	4.0Mb	4.0Mb	4.0Mb

Table 11: Uniform Distr. N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	11.8m	11.8m	11.8m	11.8m	11.8m	11.8m
	Mem:	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb
50	Sub:	0.4m	0.4m	0.4m	0.4m	0.4m	0.3m
	Ren:	11.8m	11.8m	11.8m	11.8m	11.8m	11.8m
	Mem:	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb	1.0Mb
10	Sub:	0.4m	1.5m	1.5m	1.5m	1.5m	1.5m
	Ren:	11.8m	12.2m	11.8m	11.8m	12.2m	12.2m
	Mem:	1.0Mb	10.7Mb	10.7Mb	10.7Mb	10.7Mb	10.7Mb

Table 12: Uniform Distr. N=40

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	325.5m	209.3m	128.4m	109.2m	107.5m	106.6m
	Mem:	0.3Mb	0.4Mb	0.6Mb	0.8Mb	0.8Mb	0.8Mb
50	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	323.3m	201.4m	102.2m	70.8m	65.5m	65.1m
	Mem:	0.3Mb	0.5Mb	0.7Mb	1.0Mb	1.4Mb	1.5Mb
10	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	325.1m	203.2m	101.3m	63.8m	57.2m	56.8m
	Mem:	0.3Mb	0.5Mb	0.7Mb	1.3Mb	2.7Mb	5.2Mb

Table 13: Teapot N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	56.8m	36.3m	36.3m	36.3m	36.3m	35.8m
	Mem:	0.3Mb	0.6Mb	0.6Mb	0.6Mb	0.6Mb	0.6Mb
50	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	55.9m	27.9m	27.9m	27.9m	27.9m	27.9m
	Mem:	0.3Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb
10	Sub:	0.1m	0.1m	0.2m	0.6m	0.6m	0.6m
	Ren:	56.8m	24.5m	24.5m	24.5m	24.5m	24.5m
	Mem:	0.3Mb	1.7Mb	6.5Mb	7.0Mb	7.0Mb	7.0Mb

Table 14: Teapot N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	27.9m	25.3m	25.8m	25.8m	25.8m	25.8m
	Mem:	0.3Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb	0.9Mb
50	Sub:	0.1m	0.1m	0.2m	0.1m	0.1m	0.2m
	Ren:	28.4m	23.6m	23.1m	23.6m	23.6m	23.1m
	Mem:	0.3Mb	3.4Mb	3.4Mb	3.4Mb	3.4Mb	3.4Mb
10	Sub:	0.1m	0.4m	22.2m	22.8m	22.1m	22.1m
	Ren:	28.4m	23.1m	24.5m	23.1m	23.1m	23.1m
	Mem:	0.3Mb	11.9Mb	16.1Mb	16.1Mb	16.1Mb	16.1Mb

Table 15: Teapot N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	23.1m	23.1m	23.1m	23.1m	23.1m	23.1m
	Mem:	0.5Mb	0.5Mb	0.5Mb	0.5Mb	0.5Mb	0.5Mb
50	Sub:	0.1m	0.1m	0.1m	0.1m	0.1m	0.1m
	Ren:	23.1m	23.1m	23.1m	23.1m	23.1m	23.1m
	Mem:	0.5Mb	1.8Mb	1.8Mb	1.8Mb	1.8Mb	1.8Mb
10	Sub:	0.1m	6.6m	*	*	*	*
	Ren:	23.1m	26.6m	*	*	*	*
	Mem:	0.5Mb	186.4Mb	*	*	*	*

Table 16: Teapot N=40

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	20.8m	21.6m	22.6m	22.8m	23.4m	23.6m
	Ren:	3809.8m	2789.2m	2055.2m	1649.8m	1370.1m	762.0m
	Mem:	5.0Mb	5.0Mb	5.2Mb	5.8Mb	6.8Mb	6.9Mb
50	Sub:	20.2m	22.2m	23.1m	23.4m	23.5m	23.5m
	Ren:	3781.8m	2852.1m	2104.1m	1649.8m	1433.0m	762.0m
	Mem:	5.0Mb	5.0Mb	5.2Mb	5.9Mb	7.2Mb	7.8Mb
10	Sub:	20.0m	21.4m	22.0m	22.5m	23.0m	23.0m
	Ren:	3753.9m	2796.2m	2062.2m	1679.6m	1356.2m	762.0m
	Mem:	5.0Mb	5.0Mb	5.2Mb	6.0Mb	7.5Mb	9.3Mb

Table 17: Teapot in a Temple N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	17.3m	17.6m	17.9m	17.9m	17.9m	17.8m
	Ren:	1551.9m	363.5m	55.9m	55.9m	62.9m	55.9m
	Mem:	4.9Mb	6.7Mb	8.7Mb	8.7Mb	8.7Mb	8.7Mb
50	Sub:	16.8m	17.1m	17.2m	17.2m	17.2m	17.2m
	Ren:	1558.9m	356.5m	55.9m	55.9m	55.9m	55.9m
	Mem:	4.9Mb	6.8Mb	8.9Mb	9.3Mb	9.3Mb	9.3Mb
10	Sub:	0.9m	17.1m	17.3m	18.4m	18.4m	18.1m
	Ren:	1557.9m	356.5m	55.9m	55.9m	55.9m	55.9m
	Mem:	4.9Mb	7.0Mb	13.0Mb	37.1Mb	37.1Mb	37.1Mb

Table 18: Teapot in a Temple N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	16.7m	17.4m	17.4m	17.5m	17.6m	17.7m
	Ren:	1230.3m	69.9m	69.9m	69.9m	69.9m	69.9m
	Mem:	5.1Mb	13.7Mb	14.6Mb	14.6Mb	14.6Mb	14.6Mb
50	Sub:	16.8m	17.9m	18.1m	17.9m	17.9m	17.9m
	Ren:	1244.3m	69.9m	62.9m	62.9m	62.9m	62.9m
	Mem:	5.1Mb	17.2Mb	22.1Mb	22.1Mb	22.1Mb	22.1Mb
10	Sub:	18.1m	17.6m	19.0m	19.1m	18.9m	18.9m
	Ren:	1223.3m	69.9m	48.9m	48.9m	48.9m	48.9m
	Mem:	5.1Mb	18.6Mb	54.2Mb	54.3Mb	54.3Mb	54.3Mb

Table 19: Teapot in a Temple N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	17.1m	19.2m	19.2m	18.9m	18.9m	18.8m
	Ren:	1153.4m	35.0m	35.0m	35.0m	35.0m	35.0m
	Mem:	5.7Mb	16.4Mb	16.4Mb	16.4Mb	16.4Mb	16.4Mb
50	Sub:	16.6m	27.1m	26.9m	26.9m	26.9m	27.1m
	Ren:	1104.5m	35.0m	35.0m	35.0m	35.0m	35.0m
	Mem:	5.7Mb	39.7Mb	39.7Mb	39.7Mb	39.7Mb	39.7Mb
10	Sub:	16.6m	48.5m	*	*	*	*
	Ren:	1097.5m	41.9m	*	*	*	*
	Mem:	5.7Mb	159.5Mb	*	*	*	*

Table 20: Teapot in a Temple N=40

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.4m	0.5m	0.5m	0.6m	0.6m	0.6m
	Ren:	5360.0m	3988.1m	1924.1m	837.1m	522.5m	394.9m
	Mem:	0.7Mb	0.7Mb	0.8Mb	1.1Mb	1.4Mb	1.6Mb
50	Sub:	0.4m	0.5m	0.5m	0.6m	0.6m	0.6m
	Ren:	5365.2m	3991.6m	1946.8m	833.6m	477.1m	344.3m
	Mem:	0.7Mb	0.8Mb	0.9Mb	1.2Mb	1.7Mb	2.3Mb
10	Sub:	0.4m	0.5m	0.5m	0.6m	0.6m	0.6m
	Ren:	5354.7m	4005.6m	1936.4m	828.4m	454.4m	300.6m
	Mem:	0.7Mb	0.8Mb	1.1Mb	2.0Mb	4.0Mb	7.0Mb

Table 21: Scene from Lumpy N=2

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.3m	0.3m	0.3m	0.3m	0.3m
	Ren:	1539.7m	216.7m	195.7m	194.0m	194.0m	194.0m
	Mem:	0.7Mb	1.1Mb	1.2Mb	1.2Mb	1.2Mb	1.2Mb
50	Sub:	0.3m	0.3m	0.4m	0.5m	0.4m	0.5m
	Ren:	1536.2m	131.1m	108.3m	108.3m	110.1m	110.1m
	Mem:	0.7Mb	1.5Mb	2.0Mb	2.4Mb	2.4Mb	2.4Mb
10	Sub:	0.3m	0.4m	0.9m	3.8m	4.7m	5.0m
	Ren:	1532.7m	129.3m	97.9m	101.3m	101.3m	101.3m
	Mem:	0.7Mb	2.7Mb	11.0Mb	22.8Mb	24.7Mb	25.0Mb

Table 22: Scene from Lumpy N=10

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	0.4m	0.4m	0.4m	0.4m	0.4m
	Ren:	505.1m	120.6m	118.8m	117.1m	113.6m	113.6m
	Mem:	0.8Mb	2.4Mb	2.5Mb	2.5Mb	2.5Mb	2.5Mb
50	Sub:	0.3m	0.4m	0.5m	0.5m	0.5m	0.5m
	Ren:	520.8m	103.1m	101.3m	101.3m	103.1m	106.6m
	Mem:	0.8Mb	2.7Mb	3.2Mb	3.2Mb	3.2Mb	3.2Mb
10	Sub:	0.3m	0.6m	55.3m	120.6m	138.5m	174.3m
	Ren:	512.0m	99.6m	99.6m	101.3m	99.6m	103.1m
	Mem:	0.8Mb	10.2Mb	60.3Mb	74.4Mb	78.0Mb	82.5Mb

Table 23: Scene from Lumpy N=20

MAXP		Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
100	Sub:	0.3m	2.8m	2.8m	2.8m	2.8m	2.8m
	Ren:	251.7m	96.1m	96.1m	96.1m	96.1m	94.4m
	Mem:	1.2Mb	7.3Mb	7.3Mb	7.3Mb	7.3Mb	7.3Mb
50	Sub:	0.3m	5.2m	6.9m	7.0m	7.0m	7.0m
	Ren:	246.4m	94.4m	94.4m	94.4m	94.4m	94.4m
	Mem:	1.2Mb	14.5Mb	16.8Mb	16.8Mb	16.8Mb	16.8Mb
10	Sub:	0.3m	6.7m	*	*	*	*
	Ren:	248.2m	92.6m	*	*	*	*
	Mem:	1.2Mb	45.4Mb	*	*	*	*

Table 24: Scene from Lumpy N=40

9 Conclusion

The results indicate that this hybrid of adaptive and regular spatial subdivision is useful for speeding up the rendering of typical scenes used in animation. It avoids the problems of generating object hierarchies, the excess vertical traversal of octrees, and the extreme memory requirements of regular voxel methods.

Cleary shows through a rigorous theoretical analysis that regular voxel subdivision methods can reduce ray tracing overhead to less than 130% of the minimum required to intersect every ray with one object and perform shading calculations [Cleary 88]. His analysis assumes that there is sufficient memory available to subdivide a scene to the required level, and that the objects in the scene are uniformly distributed throughout the object space. Adaptive voxel subdivision performs nearly as well as regular subdivision when the objects are uniformly distributed, and can maintain that level of performance when scenes are not so evenly constructed.

Since the speed of rendering varies with the resolution of the voxel grids and the depth of the trees, and since these vary from scene to scene, a method of determining the optimal resolution and tree depth for a given scene is required. One approach we are investigating analyses the distribution of objects in a voxel grid and determines whether a grid should be subdivided at a finer resolution, or whether certain voxels need to be recursively subdivided, and at what resolution. This is part of a research effort at the University of Calgary to develop a ray tracer that will perform well with any type of scene, and will not require hand tuning to achieve optimum performance.

10 Acknowledgements

We would like to acknowledge the help of all those who have contributed to the University of Calgary *Graphicsland* project over the years. This research is partially supported by grants from the Natural Sciences and Engineering Research Council of Canada.

References

[Amanatides 87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proc. Eurographics '87*, 1987.

[Arnaldi 87] B. Arnaldi, T. Priol, and K. Bouatouch. A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer*, 3(2):98–108, 1987.

[Arvo 87] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4), 1987.

[Cleary 88] John Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, pages 65–83, July 1988.

[Fujimoto 86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 1986.

[Glassner 84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, pages 15–22, October 1984.

[Jevans 88] David Jevans and Brian Wyvill. Ray tracing implicit surfaces. Technical Report 88/292/04, University of Calgary, 1988.

[Kaplan 85] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *SIGGRAPH '85 Course Notes 11*, 1985.

[Kay 86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Computer Graphics*, volume 20, pages 269–278. ACM SIGGRAPH, 1986.

[Pearce 87] Andrew Pearce. An implementation of ray tracing using multiprocessor and spatial subdivision. Master's thesis, University of Calgary, Dept. of Computer Science, 1987.

[Rubin 80] Steve M. Rubin and Turner Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Snyder 87] John M. Snyder and Alan H. Barr. Ray tracing complex models with surface tessellations. *Computer Graphics*, 21(4), 1987.

[Vatti 84] Reddy Vatti. Multiprocessor ray tracing. Master's thesis, University of Calgary, Dept. of Computer Science, 1984.